DA6400; Programming Assignment - 02

Rohit Kumar (DA24S003)
Sankalp Shrivastava (DA24S021)

1 Code Repository

The full source code and additional details for this assignment can be found on GitHub:
`https://github.com/DA6400-RL-JanMay2025/programming-assignment-02-da24s003_da24s021`

2 Introduction

In this assignment, we explore modern reinforcement learning (RL) algorithms by applying them to two classic control problems: `Acrobot-v1` and `CartPole-v1`, available through the Gymnasium library.

We focus on two families of algorithms: *Dueling Deep Q-Networks (Dueling-DQN)* and *Monte Carlo REINFORCE*. For each algorithm, we implement and compare two variants. In the case of Dueling-DQN, we evaluate both average and max-based advantage normalization approaches. For REINFORCE, we assess the algorithm both with and without a value function baseline. Our objective is to train these agents effectively and analyze their comparative performance in both environments.

3 Environment Descriptions

3.1 CartPole-v1 Environment

CartPole-v1 is a classic balancing problem where a pole is attached to a cart that moves along a frictionless track. The environment includes:

- A continuous state space with four variables: cart position, cart velocity, pole angle, and pole angular velocity.
- A discrete action space with two actions: move left or move right.
- A reward of $+1$ for every timestep the pole remains upright.
- The episode terminates when the pole falls beyond a threshold angle or the cart moves out of the predefined bounds.

The agent's goal is to learn a policy that applies appropriate left or right forces to keep the pole balanced for as long as possible.

3.2 Acrobot-v1 Environment

Acrobot-v1 simulates a two-link pendulum system with only the second joint actuated. The agent must learn to swing the free end of the pendulum above a certain height, starting from a downward-hanging position. The environment includes:

- A continuous state space describing the angles and angular velocities of the two links.
- A discrete action space with three actions: apply a torque of -1, 0, or +1 to the actuated joint.
- A reward of -1 per timestep until the goal is achieved, encouraging faster solutions.
- The episode ends when the tip of the pendulum reaches the required height or the time limit is reached.

This environment requires the agent to learn momentum-based strategies to reach the target efficiently.

4 Algorithms: Dueling-DQN and Monte Carlo REINFORCE

4.1 Dueling-DQN

Dueling Deep Q-Networks improve the stability and efficiency of Q-learning by decomposing the Q-function into two separate estimators:

- $V(s;\theta)$: the value function, representing the expected return from state s.
- $A(s,a;\theta)$: the advantage function, estimating the relative value of each action in state s.

We implement two variants of the dueling architecture:

1. **Type-1: Mean Normalization**

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \frac{1}{|A|} \sum_{a' \in A} A(s,a';\theta) \right)$$

2. **Type-2: Max Normalization**

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \max_{a' \in A} A(s,a';\theta) \right)$$

Both variants are evaluated in terms of learning stability and final performance across the two environments.

4.2 Monte Carlo REINFORCE

REINFORCE is a policy gradient algorithm that updates the policy parameters using complete trajectories sampled from the environment. We implement two versions:

1. **Without Baseline:**

$$\theta \leftarrow \theta + \alpha G_t \frac{\nabla \pi(A_t|S_t;\theta)}{\pi(A_t|S_t;\theta)}$$

2. **With Baseline:**

$$\theta \leftarrow \theta + \alpha(G_t - V(S_t;\Phi)) \frac{\nabla \pi(A_t|S_t;\theta)}{\pi(A_t|S_t;\theta)}$$

   The baseline value function $V(S_t;\Phi)$ is learned using the TD(0) method to reduce variance in gradient estimates.

We compare the performance of both versions in terms of convergence rate and stability in the `CartPole-v1` and `Acrobot-v1` environments.

5 Algorithm Implementation

In this section, we provide detailed descriptions of the two reinforcement learning algorithms implemented in this assignment: Dueling Deep Q-Networks (Dueling-DQN) and Monte Carlo REINFORCE. For each algorithm, we explore two variants and compare their practical impact on learning performance.

5.1 Dueling-DQN Variants

Dueling-DQN is an extension of the Deep Q-Network that improves learning efficiency by decomposing the Q-value function into two estimators: the value of being in a state $V(s)$, and the advantage of taking a specific action $A(s,a)$ in that state. This separation allows the agent to more effectively evaluate states even when the action choices do not differ much.

We implemented two variants of Dueling-DQN based on how the advantage function is normalized:

Type-1: Mean Normalization

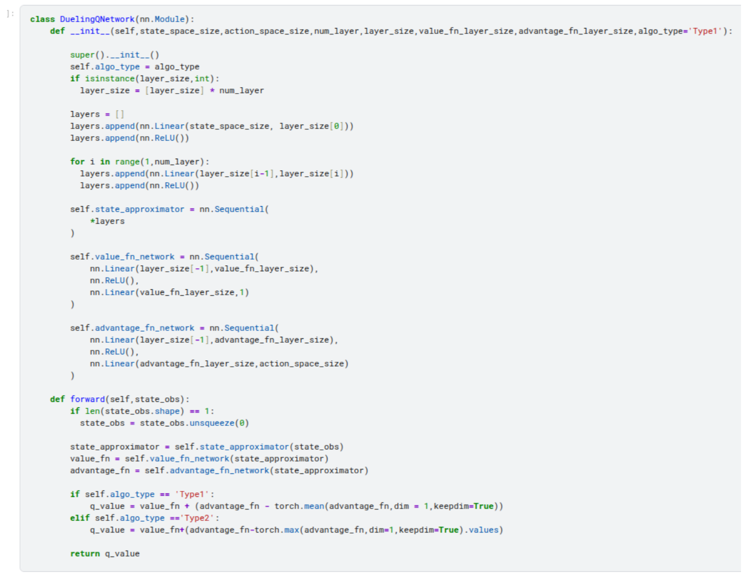
$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \frac{1}{|A|} \sum_{a' \in A} A(s,a';\theta) \right)$$

Type-2: Max Normalization

$$Q(s, a; \theta) = V(s; \theta) + \left(A(s, a; \theta) - \max_{a' \in A} A(s, a'; \theta) \right)$$

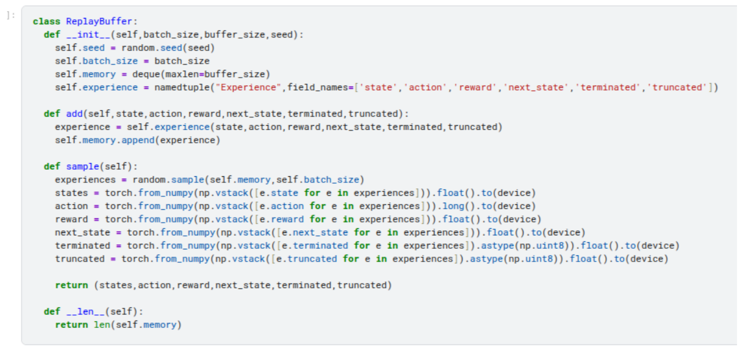
Both variants use an **epsilon-greedy** and **Softmax** strategy for exploration and were trained using experience replay and a target network for stability.

Below is a screenshot of the Dueling-DQN implementation:



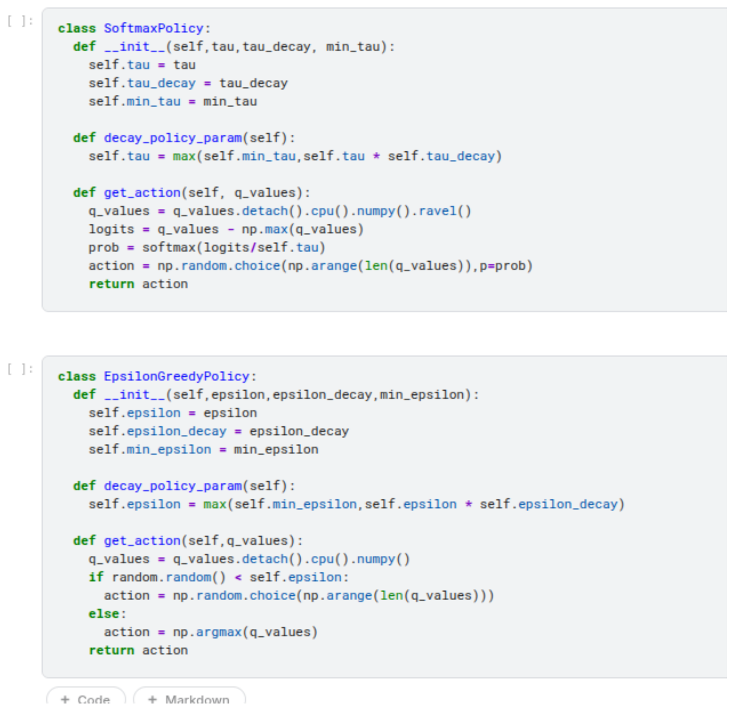
```
class DuelingQNetwork(nn.Module):
    def __init__(self,state_space_size,action_space_size,num_layer,layer_size,value_fn_layer_size,advantage_fn_layer_size,algo_type='Type1'):

        super().__init__()
        self.algo_type = algo_type
        if isinstance(layer_size,int):
          layer_size = [layer_size] * num_layer

        layers = []
        layers.append(nn.Linear(state_space_size, layer_size[0]))
        layers.append(nn.ReLU())

        for i in range(1,num_layer):
          layers.append(nn.Linear(layer_size[i-1],layer_size[i]))
          layers.append(nn.ReLU())

        self.state_approximator = nn.Sequential(
            *layers
        )

        self.value_fn_network = nn.Sequential(
            nn.Linear(layer_size[-1],value_fn_layer_size),
            nn.ReLU(),
            nn.Linear(value_fn_layer_size,1)
        )

        self.advantage_fn_network = nn.Sequential(
            nn.Linear(layer_size[-1],advantage_fn_layer_size),
            nn.ReLU(),
            nn.Linear(advantage_fn_layer_size,action_space_size)
        )

    def forward(self,state_obs):
        if len(state_obs.shape) == 1:
          state_obs = state_obs.unsqueeze(0)

        state_approximator = self.state_approximator(state_obs)
        value_fn = self.value_fn_network(state_approximator)
        advantage_fn = self.advantage_fn_network(state_approximator)

        if self.algo_type == 'Type1':
            q_value = value_fn + (advantage_fn - torch.mean(advantage_fn,dim = 1,keepdim=True))
        elif self.algo_type =='Type2':
            q_value = value_fn+(advantage_fn-torch.max(advantage_fn,dim=1,keepdim=True).values)

        return q_value
```

Figure 1: Implementation of Dueling-DQN



```
class ReplayBuffer:
  def __init__(self,batch_size,buffer_size,seed):
    self.seed = random.seed(seed)
    self.batch_size = batch_size
    self.memory = deque(maxlen=buffer_size)
    self.experience = namedtuple("Experience",field_names=['state','action','reward','next_state','terminated','truncated'])

  def add(self,state,action,reward,next_state,terminated,truncated):
    experience = self.experience(state,action,reward,next_state,terminated,truncated)
    self.memory.append(experience)

  def sample(self):
    experiences = random.sample(self.memory,self.batch_size)
    states = torch.from_numpy(np.vstack([e.state for e in experiences])).float().to(device)
    action = torch.from_numpy(np.vstack([e.action for e in experiences])).long().to(device)
    reward = torch.from_numpy(np.vstack([e.reward for e in experiences])).float().to(device)
    next_state = torch.from_numpy(np.vstack([e.next_state for e in experiences])).float().to(device)
    terminated = torch.from_numpy(np.vstack([e.terminated for e in experiences]).astype(np.uint8)).float().to(device)
    truncated = torch.from_numpy(np.vstack([e.truncated for e in experiences]).astype(np.uint8)).float().to(device)

    return (states,action,reward,next_state,terminated,truncated)

  def __len__(self):
    return len(self.memory)
```

Figure 2: Implementation of Replay Buffer



```
class SoftmaxPolicy:
  def __init__(self,tau,tau_decay, min_tau):
    self.tau = tau
    self.tau_decay = tau_decay
    self.min_tau = min_tau

  def decay_policy_param(self):
    self.tau = max(self.min_tau,self.tau * self.tau_decay)

  def get_action(self, q_values):
    q_values = q_values.detach().cpu().numpy().ravel()
    logits = q_values - np.max(q_values)
    prob = softmax(logits/self.tau)
    action = np.random.choice(np.arange(len(q_values)),p=prob)
    return action
```

```
class EpsilonGreedyPolicy:
  def __init__(self,epsilon,epsilon_decay,min_epsilon):
    self.epsilon = epsilon
    self.epsilon_decay = epsilon_decay
    self.min_epsilon = min_epsilon

  def decay_policy_param(self):
    self.epsilon = max(self.min_epsilon,self.epsilon * self.epsilon_decay)

  def get_action(self,q_values):
    q_values = q_values.detach().cpu().numpy()
    if random.random() < self.epsilon:
      action = np.random.choice(np.arange(len(q_values)))
    else:
      action = np.argmax(q_values)
    return action
```

Figure 3: Implementation of Helper Functions

5.2 Monte Carlo REINFORCE Variants

Monte Carlo REINFORCE is a classic policy gradient algorithm that updates the policy parameters based on entire episodes sampled from the environment. It aims to maximize the expected return by following the gradient of the performance objective.

Without Baseline In this variant, the policy is updated using the raw return from each trajectory:

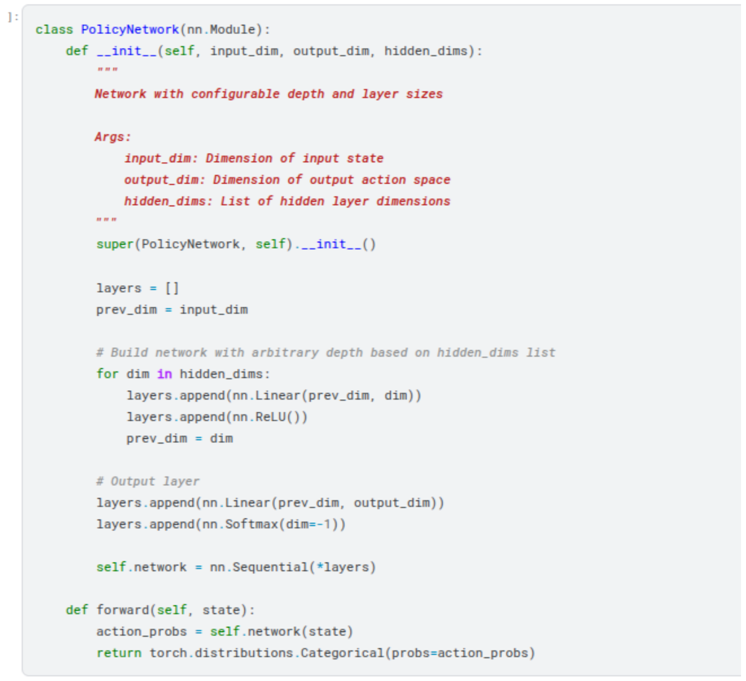
$$\theta \leftarrow \theta + \alpha G_t \frac{\nabla \pi(A_t|S_t;\theta)}{\pi(A_t|S_t;\theta)}$$

This approach is unbiased but can suffer from high variance.

With Baseline To reduce variance in policy updates, a value function baseline $V(S_t;\Phi)$ is introduced. This value function is updated using the TD(0) method, and the policy update becomes:

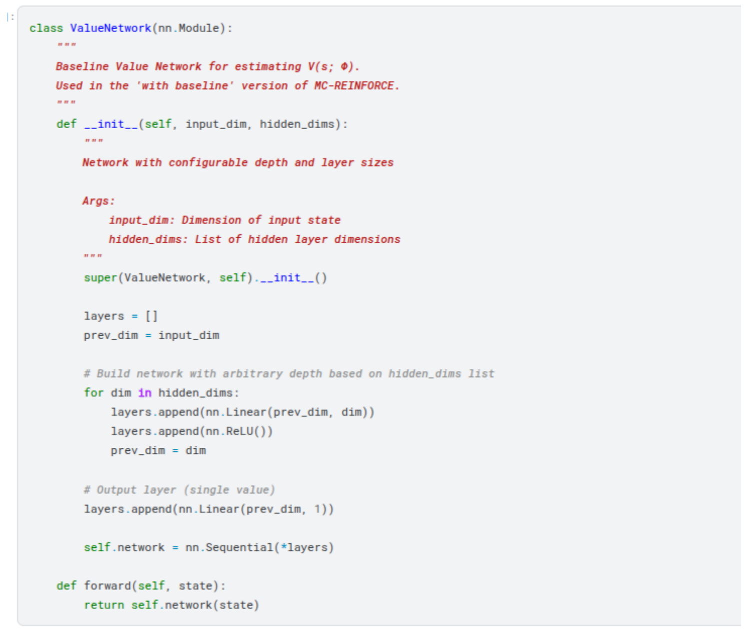
$$\theta \leftarrow \theta + \alpha(G_t - V(S_t;\Phi)) \frac{\nabla \pi(A_t|S_t;\theta)}{\pi(A_t|S_t;\theta)}$$

Below is a screenshot of the REINFORCE implementation with and without baseline:



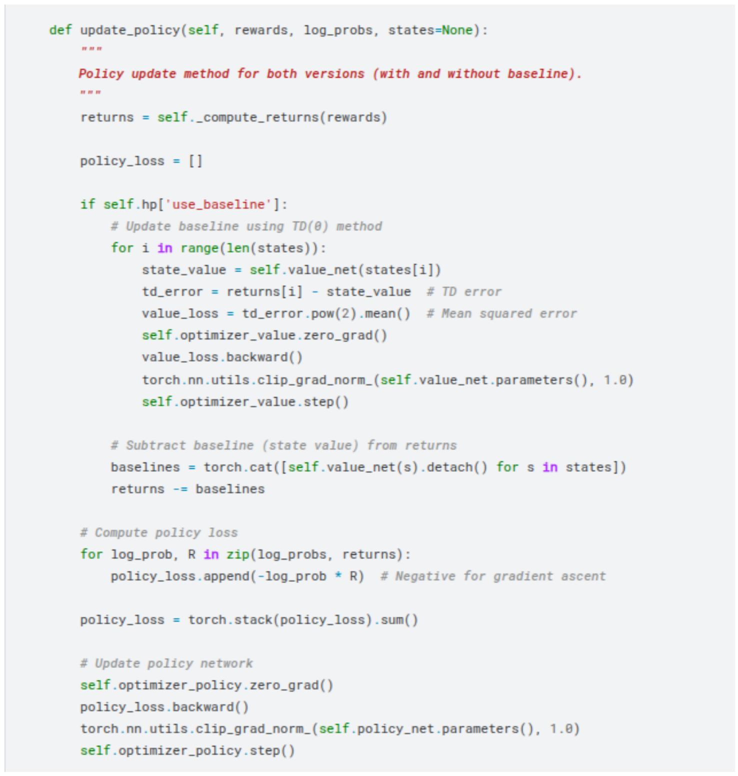
```
]:
class PolicyNetwork(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dims):
        """
        Network with configurable depth and layer sizes

        Args:
            input_dim: Dimension of input state
            output_dim: Dimension of output action space
            hidden_dims: List of hidden layer dimensions
        """
        super(PolicyNetwork, self).__init__()

        layers = []
        prev_dim = input_dim

        # Build network with arbitrary depth based on hidden_dims list
        for dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, dim))
            layers.append(nn.ReLU())
            prev_dim = dim

        # Output layer
        layers.append(nn.Linear(prev_dim, output_dim))
        layers.append(nn.Softmax(dim=-1))

        self.network = nn.Sequential(*layers)

    def forward(self, state):
        action_probs = self.network(state)
        return torch.distributions.Categorical(probs=action_probs)
```

Figure 4: Implementation of Policy Network for MC Reinforce



```
class ValueNetwork(nn.Module):
    """
    Baseline Value Network for estimating V(s; Φ).
    Used in the 'with baseline' version of MC-REINFORCE.
    """
    def __init__(self, input_dim, hidden_dims):
        """
        Network with configurable depth and layer sizes

        Args:
            input_dim: Dimension of input state
            hidden_dims: List of hidden layer dimensions
        """
        super(ValueNetwork, self).__init__()

        layers = []
        prev_dim = input_dim

        # Build network with arbitrary depth based on hidden_dims list
        for dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, dim))
            layers.append(nn.ReLU())
            prev_dim = dim

        # Output layer (single value)
        layers.append(nn.Linear(prev_dim, 1))

        self.network = nn.Sequential(*layers)

    def forward(self, state):
        return self.network(state)
```

Figure 5: Implementation of Value Network for MC Reinforce, baseline version



```
def update_policy(self, rewards, log_probs, states=None):
    """
    Policy update method for both versions (with and without baseline).
    """
    returns = self._compute_returns(rewards)

    policy_loss = []

    if self.hp['use_baseline']:
        # Update baseline using TD(0) method
        for i in range(len(states)):
            state_value = self.value_net(states[i])
            td_error = returns[i] - state_value  # TD error
            value_loss = td_error.pow(2).mean()  # Mean squared error
            self.optimizer_value.zero_grad()
            value_loss.backward()
            torch.nn.utils.clip_grad_norm_(self.value_net.parameters(), 1.0)
            self.optimizer_value.step()

        # Subtract baseline (state value) from returns
        baselines = torch.cat([self.value_net(s).detach() for s in states])
        returns -= baselines

    # Compute policy loss
    for log_prob, R in zip(log_probs, returns):
        policy_loss.append(-log_prob * R)  # Negative for gradient ascent

    policy_loss = torch.stack(policy_loss).sum()

    # Update policy network
    self.optimizer_policy.zero_grad()
    policy_loss.backward()
    torch.nn.utils.clip_grad_norm_(self.policy_net.parameters(), 1.0)
    self.optimizer_policy.step()
```

Figure 6: Implementation of Update Policy

6 Experiments and Analysis

6.1 Overview

This section presents the reward curves for two reinforcement learning algorithms listed below, evaluated on `Acrobot-v1` and `CartPole-v1` environments:

- **Dueling Deep Q-Network (Dueling DQN)**
- **Monte Carlo REINFORCE** (with and without a baseline)

Each agent was trained for up to **1000 episodes**, and performance was monitored using *reward per episode* as the metric.

6.2 CartPole-v1: Dueling DQN Analysis

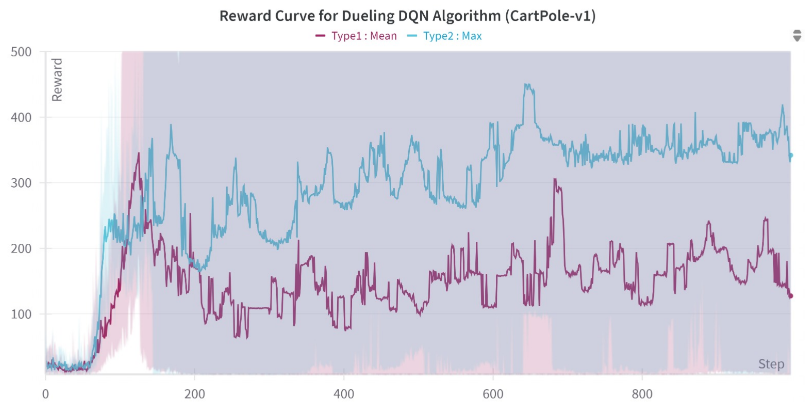


Figure 7: Reward Curve for Dueling DQN on CartPole-v1 (Type-1: Mean, Type-2: Max)

Performance of the Dueling DQN algorithm using two different Q-value estimation strategies in the CartPole-v1 environment. The Type-2 variant (Max aggregation) shows significantly better and more stable performance, steadily reaching higher rewards, peaking around 450 and maintaining above 300 for the majority of training.

On the other hand, the Type-1 variant (Mean aggregation) struggles with consistency. While it initially improves and occasionally reaches spikes of over 300, the rewards often collapse and fluctuate around the 100–200 range, indicating instability or ineffective value estimation.

This suggests that Type-2's use of the max advantage provides stronger gradients for learning and better action distinction, especially in simpler environments like CartPole where decisive policies can emerge quickly.

- **Best Configuration**
  - `advantage_fn_layer_size`: 32
  - `batch_size`: 128
  - `env_name`: "CartPole-v1"
  - `experiment`: "DuelingDQN"
  - `exploration_policy`: "Softmax"
  - `gamma`: 0.998
  - `layer_size`: [128,64]
  - `learning_rate`: 0.0001
  - `num_layer`: 2
  - `param_decay`: 0.9995
  - `policy_param`: 1
  - `policy_param_min_value`: 0.1
  - `target_network_replacement_freq`: 20
  - `type`: "Type1(Mean)"
  - `value_fn_layer_size`: 32
- **Best Configuration**
  - `advantage_fn_layer_size`: 32
  - `batch_size`: 128
  - `env_name`: "CartPole-v1"
  - `experiment`: "DuelingDQN"
  - `exploration_policy`: "Softmax"
  - `gamma`: 0.998
  - `layer_size`: [128,64]
  - `learning_rate`: 0.0001
  - `num_layer`: 2
  - `param_decay`: 0.995
  - `policy_param`: 1
  - `policy_param_min_value`: 0.01
  - `target_network_replacement_freq`: 50
  - `type`: "Type2(Max)"
  - `value_fn_layer_size`: 32

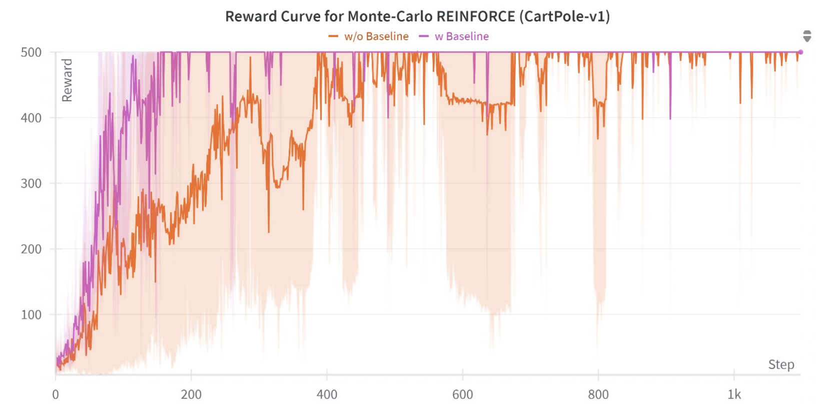


Figure 8: Reward Curve for Monte-Carlo REINFORCE on CartPole-v1 (with and without Baseline)

6.3 CartPole-v1: Monte-Carlo REINFORCE Analysis

The figure showcases the performance of Monte-Carlo REINFORCE on the CartPole-v1 environment, comparing learning curves with and without a baseline.

The use of a baseline (purple curve) leads to faster and more stable convergence. It reaches near-optimal performance (reward = 500) within the first 150 episodes, and remains consistent thereafter. This demonstrates the effectiveness of baseline-based variance reduction in policy gradient methods.

Without a baseline (orange curve), learning is noticeably slower and exhibits high variance throughout training. Although it eventually converges to high-reward episodes, the instability—evident from the wide shaded region and frequent reward drops—illustrates how sensitive vanilla REINFORCE is to high variance in return estimates.

This experiment confirms that incorporating a baseline significantly enhances both sample efficiency and stability in policy-gradient-based reinforcement learning.

- **Best Configuration**
  - `baseline`: False
  - `env_name`: "CartPole-v1"
  - `experiment`: "MCReinforce"
  - `gamma`: 0.99
  - `layer_size`: [64]
  - `learning_rate`: 0.01
  - `lr_decay`:1

  - `lr_value`: 0.001
  - `n_episodes`: 1000

- **Best Configuration**
  - `baseline`: True
  - `env_name`: "CartPole-v1"
  - `experiment`: "MCReinforce"
  - `gamma`: 0.99
  - `layer_size`: [64]
  - `learning_rate`: 0.01
  - `lr_decay`: 1
  - `lr_value`: 0.001
  - `n_episodes`: 1000

6.4 Acrobot-v1: Dueling DQN Analysis

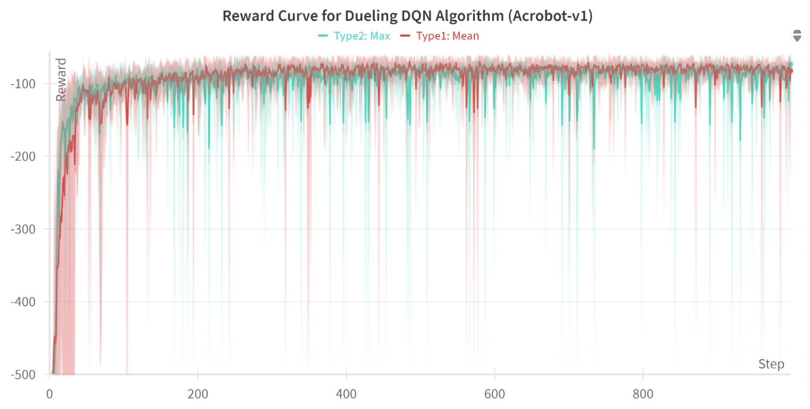


Figure 9: Reward Curve for Dueling DQN on Acrobot-v1 (Mean vs Max)

The above plot evaluates the performance of the Dueling Deep Q-Network (DQN) on the Acrobot-v1 environment, with a comparison between the mean and maximum episodic rewards.

Both curves demonstrate rapid improvement in the initial phase, with rewards climbing from around -500 to the optimal region near -100. The mean reward curve (red) shows a smoother trajectory and converges relatively early, indicating that the average performance becomes consistently strong after about 100 episodes.

The maximum reward (cyan) follows a similar trend but with slightly higher variance, especially in early training. This is expected, as maximum rewards can spike due to occasional lucky episodes even while the policy is still stabilizing.

Overall, Dueling DQN demonstrates stable and reliable learning in Acrobot-v1. The architecture's ability to separate value and advantage functions likely aids in learning effective state representations, especially in environments with sparse reward signals.

- **Best Configuration**
  - `advantage_fn_layer_size`: 32
  - `batch_size`: 128
  - `env_name`: "CartPole-v1"
  - `experiment`: "DuelingDQN"
  - `exploration_policy`: "Softmax"
  - `gamma`: 0.998
  - `layer_size`: [128,64]
  - `learning_rate`: 0.001
  - `num_layer`: 2
  - `param_decay`: 0.995
  - `policy_param`: 1
  - `policy_param_min_value`: 0.01
  - `target_network_replacement_freq`: 50
  - `type`: "Type1(Mean)"
  - `value_fn_layer_size`: 32
- **Best Configuration**
  - `advantage_fn_layer_size`: 32
  - `batch_size`: 128
  - `env_name`: "Acrobot-v1"
  - `experiment`: "DuelingDQN"
  - `exploration_policy`: "Softmax"
  - `gamma`: 0.998
  - `layer_size`: [128,64]
  - `learning_rate`: 0.0001
  - `num_layer`: 2
  - `param_decay`: 0.995
  - `policy_param`: 1
  - `policy_param_min_value`: 0.01
  - `target_network_replacement_freq`: 50
  - `type`: "Type2(Max)"
  - `value_fn_layer_size`: 32

6.5 Acrobot-v1: Monte Carlo REINFORCE Analysis

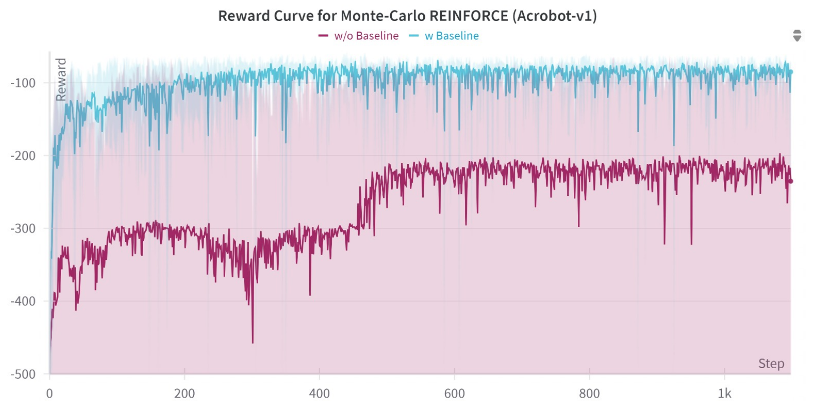


Figure 10: Reward Curve for Monte Carlo REINFORCE on Acrobot-v1 (w/ and w/o Baseline)

This plot compares the learning performance of the Monte Carlo REINFORCE algorithm with and without the use of a baseline in the `Acrobot-v1` environment. The blue curve, representing REINFORCE with a baseline, demonstrates a smoother and significantly more effective learning trajectory. Rewards improve quickly and stabilize around -100, which is near the optimal return in this environment.

In contrast, the baseline-free variant (shown in maroon) exhibits high variance, slower improvement, and struggles to surpass the

reward mark consistently. Large dips and instability suggest that the absence of a variance-reducing baseline makes training noisy and inefficient.

The results clearly validate that including a baseline in policy gradient methods helps reduce variance in gradient estimation, which is especially beneficial in environments with sparse or delayed rewards like Acrobot.

- **Best Configuration**
  - `baseline`: False
  - `env_name`: "Acrobot-v1"
  - `experiment`: "MCReinforce"
  - `gamma`: 0.99
  - `layer_size`: [64]
  - `learning_rate`: 0.01
  - `lr_decay`:0.995
  - `lr_value`: 0.001

  - `n_episodes`: 1000

- **Best Configuration**
  - `baseline`: True
  - `env_name`: "Acrobot-v1"
  - `experiment`: "MCReinforce"
  - `gamma`: 0.99
  - `layer_size`: [128]
  - `learning_rate`: 0.01
  - `lr_decay`: 0.995
  - `lr_value`: 0.001
  - `n_episodes`: 1000

6.6 Key Findings

Monte Carlo REINFORCE:

- Works well on dense-reward environments like `CartPole`.
- Adding a baseline significantly improves stability and sample efficiency.
- Without a baseline, training is noisy and inefficient, though it eventually improves.

Dueling DQN:

- Suitable for sparse-reward environments like `Acrobot`.
- Its architectural enhancement enables the agent to learn which states are valuable even when individual action values are similar.
- Very Unstable During Training and Very sensitive to the Hyperparameters.